VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Normalizace XQuery dotazů

XQuery normalization

2010

Bc. Filip Břuska

Zadání

Téma:

Normalizace XQuery dotazů

Zásady pro vypracování:

XQuery jazyk slouží k dotazování XML dat. Jeho poměrně volná definice umožňuje napsat stejný dotaz mnoha ekvivalentními způsoby. Tato práce by se měla zabývat převodem XQuery dotazů do normalizované formy, kde budou mít ekvivalentní zápisy XQuery dotazu vždy stejnou podobu.

1. Seznámení se s teorií a metodami normalizace XQuery dotazů.
2. Realizace vybraných metod ve vybraném programovacím jazyce. Výsledná aplikace bude mít následující vlastnosti::
   - Vstupem bude XQuery dotaz a na výstupu bude ekvivalentní normalizovaný XQuery dotaz.
   - Při zadání nepovinného parametru budou do pomocného souboru ukládány informace o provedených transformacích XQuery dotazu. Tyto informace by měly být srozumitelně popsány v samotné práci.
3. Sestavení množiny XQuery dotazů, které budou vhodně demonstrovat fungování celé aplikace.

Vedoucí:

Ing. Radim Bača, Ph.D.

Obor:

2612T025 Informatika a výpočetní technika

Akademický rok:

2009/2010

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 3. května 2010 .....

Abstrakt

Tato práce se zabývá jazykem XQuery a normalizací formulí tímto jazykem vytvořených. Jazyk XQuery slouží k vytváření dotazovacích formulí nad XML daty. Jelikož je definice tohoto jazyka poměrně volná, lze vytvořit stejný dotaz mnoha ekvivalentními způsoby. Proto se používá normalizace, která spočívá v převodu původního XQuery výrazu na výraz jazyka XQuery Core, využívajícího XQuery Core gramatiku. Tato gramatika je podmnožinou XQuery gramatiky a neobsahuje již redundantní pravidla. Cílem této práce je tedy nastudovat a implementovat algoritmus, který zajistí překlad XQuery dotazů do normalizované formy, přičemž bude platit, že ekvivalentní zápisy dotazů budou mít vždy stejnou podobu.

Klíčová slova: Diplomová práce, XML, XQuery, Gramatika, Parser, Normalizace XQuery, JavaCC, JJTree

Abstract

This diploma thesis concerns with XQuery language and with normalization of formulas created bythis language. XQuery language is used for creation of queries for XML data. Because the definition of the language is rather freely defined, a query can be created by several equivalent manners. This is the reason why normalization is used. It is based in transfer of the original XQuery expression to an expression in XQuery Core language that uses XQuery core grammar. This grammar is a subset of XQuery grammar and it does not contain redundancy rules anymore. Aim of this thesis is to study and implement an algorithm that assures XQuery queries translation into the normalized form, whereas the rule saying that the equivalent formulas are formed always the same way is kept.

Keywords: Graduation thesis, XML, XQuery, Grammar, Parser, Normalization of XQuery, JavaCC, JJTree

Seznam použitých zkratek a symbolů

XML	–	eXtensible Markup Language
XQuery	–	XML Query Language
JavaCC	–	Java Compiler Compiler
IDE	–	Integrated Development Environment

Obsah

1 Úvod **7**

2 XML **8**
2.1 Úvod o XML 8
2.2 XML technologie 9

3 XQuery **13**
3.1 Užití XQuery 13
3.2 Vlastnosti XQuery 13
3.3 Výhody XQuery 13
3.4 Rozdíl oproti XPath 14
3.5 XQuery Update Facility 14
3.6 FLWOR konstrukce 15

4 Normalizace XQuery **17**
4.1 Důvody normalizace 17
4.2 Seznámení s normalizačními pravidly 17
4.3 FLWOR normalizace 18
4.4 Ostatní normalizace 21

5 Lexikální a syntaktická analýza **25**
5.1 Využití parser generátoru 25
5.2 JavaCC 26
5.3 JJTree 27
5.4 Výhody JavaCC a JJTree 28
5.5 Rekurzivní sestup 29

6 Implementace **30**
6.1 Implementační prostředky 30
6.2 Struktura implementace 31
6.3 Implementace normalizačních pravidel 36
6.4 Generování výstupu 41
6.5 Implementovaná pravidla 42

7 Analyzování výstupních dat **43**
7.1 Normalizování redundantních výrazů 43
7.2 Detaily normalizace 43

8 Závěr **45**

9 Literatura **46**

Přílohy **47**

A Obsluha programu **48**

B Programová specifikace **49**
B.1 Vývojové prostředí 49
B.2 Latex 49
B.3 Diagramy 49

Seznam tabulek

1 Lexikální analýza pro vstup sum=4*3+2; 26

Seznam obrázků

1 Ukázka os používaných u XPath a XQuery 11
2 Normalizační pravidlo FOR 18
3 Normalizační pravidlo LET 19
4 Normalizační pravidlo WHERE 20
5 Zjednodušené normalizační pravidlo ORDER BY 20
6 Normalizační pravidlo RelativePathExpr 22
7 Normalizační pravidlo following a following-sibling 23
8 Normalizační pravidlo preceding a preceding-sibling 23
9 Zkrácené zápisy 23
10 Normalizační pravidlo If-Then-Else 24
11 Syntaktický strom 27
12 Implementační proces 32
13 Model výrazu 36
14 Normalizovaný výraz **for $a in /book, $b in /emp, $c in /salary where ...** . 37
15 WHERE transformace na IF 39
16 Transformace osy 40

Seznam výpisů zdrojového kódu

1 Ukázka XML dokumentu . . . 8
2 XPath s predikáty . . . 10
3 XPath s využitím osy a zástupného znaku . . . 11
4 XQuery Update . . . 14
5 FLWOR gramatické pravidlo . . . 15
6 Zbývající gramatická pravidla pro FLWOR konstrukce . . . 15
7 Ukázka FLWOR dotazu . . . 16
8 Funkce fs:to . . . 17
9 Původní FLWOR konstrukce obsahující FOR konstrukty . . . 18
10 Normalizována FLWOR konstrukce obsahující FOR konstrukty . . . 19
11 Původní FLWOR konstrukce . . . 21
12 Normalizovaná FLWOR konstrukce . . . 21
13 Ukázka gramatiky generující syntaktický strom . . . 26
14 Ukázka definování gramatického pravidla v JJTree . . . 28
15 Ukázka skrytého gramatického pravidla FLWORExpr v JJTree . . . 33
16 Normalizovaný výstup XQuery výrazu . . . 37
17 Normalizace redundantních XQuery výrazů . . . 43
18 Výstupní normalizovaný výraz . . . 43
19 Transformační výpis v souboru **log.txt** . . . 43
20 Podrobný transformační výpis v souboru **log.txt** pro FLWOR konstrukci . . . 44

Seznam algoritmů

1 Odstranění dětského uzlu 34
2 Prefixový průchod 37
3 FOR a LET normalizace 38
4 WHERE normalizace 39
5 Transformace os 41
6 Generování výpisu pro konstrukt FOR 41

1 Úvod

Tato práce se zabývá aplikováním normalizačních pravidel nad XQuery výrazy. Samotný jazyk XQuery poskytuje mnoho užitečných funkcí, díky kterým lze vytvořit výraz jednodušeji jak pro zapsání, tak pro pochopení. Na druhou stranu jsou však tyto výrazy redundantní. Například složený konstrukt FOR může být přepsán jako kompozice několika jednoduchých FOR konstruktů. Jazyk, který obsahuje pouze tyto jednoduché konstrukty, je nazván XQuery Core a je popsán XQuery Core gramatikou, která je podmnožinou původní XQuery gramatiky. Převedení původního XQuery výrazu do výrazu využívajícího XQuery Core gramatiku nazýváme normalizace. Zpracování XQuery výrazů provádí XQuery procesor. Přičemž normalizace je jedna z fází tohoto procesoru, která eliminuje redundance.
V této práci je implementována část normalizačních pravidel zaměřená na normalizování FLWOR konstruktů, které tvoří jádro jazyka XQuery. Abychom mohli zadaný výraz normalizovat, potřebujeme si jej nejprve převést do určité struktury. Pro reprezentování výrazu jsme zvolili stromovou strukturu, takže vstupní výraz je reprezentován stromem, ve kterém každý uzel koresponduje s určitým terminálem ze sady XQuery gramatických pravidel. Normalizace využívá rekurzivní průchod stromem. Tento způsob procházení umožňuje normalizovat jakýkoli výraz jazyka XQuery na výraz jazyka XQuery Core.
Přínosem pro mou osobu vidím v hlubším seznámení s jazykem XQuery a jeho normalizačními pravidly. Mezi další přínosy patří práce s formální gramatikou a s generátory parserů, konkrétně JavaCC a JJTree, v neposlední řadě práce se stromovou strukturou v programovacím jazyce Java.
Úvodní část práce je zaměřena na využití a hlavní výhody jazyka XQuery. Následuje vysvětlení důvodů normalizace a ukázka několika normalizačních pravidel, které jsou implementovány v této práci. V další kapitole je rozepsáno zpracování vstupního výrazu do stromové struktury. U tohoto převodu jsme využili generátory parserů JavaCC a JJTree, které využívají oficiální gramatiku pro XQuery. Navazuje část zabývající se způsobem transformování stromové struktury. Tyto transformace jsou prováděny podle normalizačních pravidel. Poslední dvě kapitoly se zaobírají použitými technologiemi a analyzováním podrobného výpisu, ve kterém jsou rozepsány všechny kroky normalizace.

2 XML

XML (eXtensible Markup Language, česky rozšířitelný značkovací jazyk) je obecně označován za značkovací jazyk. Jedná se vlastně o metajazyk, ve kterém značky popisují obsah a dopomáhají tak k jeho strukturalizaci [1, 2].

2.1 Úvod o XML

Tato práce pojednává o normalizaci jazyka XQuery, který je určen pro dotazování nad určitou kolekcí XML dat. Proto bychom si také měli přiblížit samotné XML. Tento jazyk byl vyvinut a standardizován konsorciem W3C. Umožňuje snadné vytváření konkrétních značkovacích jazyků pro různé účely a široké spektrum různých typů dat [3]. Následuje XML dokument určený pro zápis kuchařských receptů.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Poznamka je nutné přidat více receptů. -->
<recepty>
  <recept id="1">
    <nazev_receptu>"Krokant"</nazev_receptu>
    <prisady>
      <prisada mnozstvi ="4" jednotka = "lzice">voda</prisada>
      <prisada mnozstvi ="10" jednotka = "lzice">pískový cukr</prisada>
      <prisada mnozstvi ="100" jednotka = "gram">sekané ořechy</prisada>
      <prisada mnozstvi ="80" jednotka = "gram">loupané mandle</prisada>
    </prisady>
    <postup>
    Cukr svaříme s vodou na prudkém ohni, až hmota zlátne. Přisypeme sekané ořechy a
        mícháme, až jsou jím obaleny. Potom je vyklopíme na plech vytřený olejem a po
        vychladnutí roztlučeme válečkem na drobné kousky. Používáme na dorty, řezy, poháry,
        atd.
    </postup>
  </recept>
</recepty>
```

Výpis 1: Ukázka XML dokumentu

2.1.1 Proč XML?

Následující seznam obsahuje hlavní výhody použití XML:

- Jazyk je určen především pro výměnu dat mezi aplikacemi a pro publikování dokumentů.
- Standardní formát pro výměnu informací - V dnešní době existuje několik velmi rozšířených formátů dokumentů, které vyžadují speciální programy pro práci s těmito typy dokumentů. Jedná se například o programy Word nebo Excel a jejich formáty DOC případně XLS. Navíc je používána celá řada operačních a informačních systémů a není záruka, že každý uživatel vlastní příslušný software.

Je tedy potřeba používat nějaký jednoduchý otevřený formát, který není úzce svázán s nějakou platformou nebo technologií. Tím může být právě XML, který je založen na jednoduchém textu a je zpracovatelný (v případě potřeby) libovolným textovým editorem.

- Mezinárodní podpora - XML hned od samého počátku myslel na potřeby i jiných jazyků než je angličtina. Jako znaková sada se implicitně používá ISO 10646 (součástí je UTF-8)
- Vysoký informační obsah - Pomocí XML značek (tagů) vyznačujeme v dokumentu význam jednotlivých částí textu. Dokumenty tak obsahují více informací, než kdyby se používalo značkovaní zaměřené na prezentaci (vzhled) – definice písma, odsazení a podobně. XML dokumenty jsou informačně bohatší.
- Jednoduchá kontrola validace
- Nezávislost na platformě - Možnost využití mezi různorodými systémy.
- Univerzálnost - V dokumentech lze vytvářet libovolné značky. Tyto značky můžeme zanořovat. Tímto zanořováním se dostaneme od obecných značek, například knihkupectví, až k přesným informacím, jako je příjmení autora dané knihy prodávané v tomto knihkupectví.

2.1.2 Syntaxe XML

Základní syntaktická pravidla pro psaní XML dokumentu:

- Jména elementů v XML rozlišují malá a velká písmena.
- Musí mít právě jeden kořenový (root) element.
- Neprázdné elementy musí být ohraničeny startovací a ukončovací značkou. Prázdné elementy mohou být označeny tagem „prázdný element“.
- Všechny hodnoty atributů musí být uzavřeny v uvozovkách – jednoduchých (') nebo dvojitých ("), ale jednoduchá uvozovka musí být uzavřena jednoduchou a dvojitá dvojitou. Opačný pár uvozovek může být použit uvnitř hodnot.
- Elementy mohou být vnořeny, ale nemohou se překrývat; to znamená, že každý (ne kořenový) element musí být celý obsažen v jiném elementu.

2.2 XML technologie

Strukturovaný XML dokument se hodí pro ukládání dat. Abychom mohli s takto uloženými daty dále pracovat, musíme využít dostupné XML technologie. Pokud chceme definovat přesnou strukturu XML nebo se chceme dotazovat na jednotlivé části XML, případně chceme XML transformovat, tak můžeme využít následující technologie. Ve výčtu technologií chybí dotazovací jazyk XQuery, který je podrobně popsán v následující kapitole.

2.2.1 XML Schema

Pomocí XML Schema (přípona *.xsd) specifikujeme pravidla pro strukturu XML dokumentu. Pokud dokument vyhovuje definované struktuře, tak se jedná o validní XML dokument [4].

Hlavní využití XML Schema Nejdůležitější uplatnění jsou rozepsána v následujícím seznamu:

- Definování přípustné struktury dokumentu.
- Ověření správnosti dat.
- Konverze dat mezi různými datovými typy.
- Práce s daty uloženými v databázi.

2.2.2 XPath

XPath (XML Path Language) je počítačový jazyk, pomocí kterého lze adresovat části XML dokumentu. Pomocí tohoto jazyka lze z XML dokumentu vybírat jednotlivé elementy a pracovat s jejich hodnotami a atributy. Mezi další vymoženosti jazyka XPath slouží funkce, predikáty a zástupné znaky [5, 6, 7].

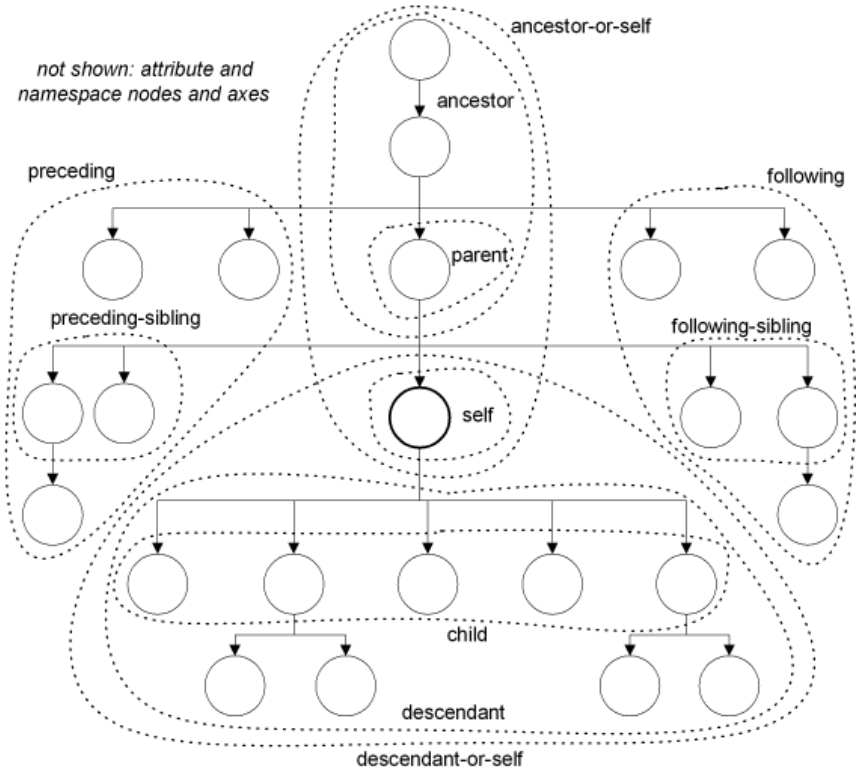
Zápis cesty Základní součástí jazyka je výraz popisující cestu. Taková cesta je tvořena posloupností přechodů mezi jednotlivými sadami uzlů, oddělených lomítky. Například pomocí příkazu **/recepty/recept/*** vypíšeme všechny elementy všech receptů. Cesta může být také zadána pomocí os. Ukázka os je na následujícím obrázku.

Predikáty Výraz v hranatých závorkách může specifikovat podmínky. Případně zde může být použito pouze číslo, které určuje pozici elementu ve vybraném souboru elementů. Atributy jsou specifikovány pomocí prefixu @.

```
//zbozi[@sleva >= @cena div 2]
```

Výpis 2: XPath s predikáty

Tento XPath výraz vypíše všechny elementy zboží, jejichž atribut sleva je nejméně polovina z ceny.



Obrázek 1: Ukázka os používaných u XPath a XQuery

Funkce Další velmi užitečnou vlastností jazyka XPath je možnost využití funkcí. Zde je seznam těch nejpoužívanějších.

- last() - Vybírá poslední element ve výběru.
- count()- Vrací počet vybraných elementů.
- string-length() - Vrací délku daného řetězce.
- name() - Vrací jméno elementu.

Zástupné znaky Také známé pod svým anglickým názvem wildcard. Jedná se o speciální znak, který zastupuje určitou část v XML struktuře. Nejčastěji používaný zástupný znak je *, který zastupuje všechny dětské elementy jistého elementu.

```
/a/b/following-sibling::*[1]
```

Výpis 3: XPath s využitím osy a zástupného znaku

Tento XPath výraz vypíše všechny elementy, které jsou prvním elementem následujícím po elementu **b**, který je potomek elementu **a**.

2.2.3 XSLT

Tato technologie (XSL Transformations) slouží k převodům XML dat do jakéhokoli jiného formátu [8]. Tento úkon provádí XSLT procesor, například **Saxon**. Transformace XSLT probíhá tak, že vybereme část dokumentu pomocí XPath a na tuto část použijeme jednu nebo více předdefinovaných šablon. V případě, že je příslušná část nalezena, XSLT provede transformaci odpovídající části zdrojového dokumentu do výstupního dokumentu. Tento XSLT transformační proces tedy používá deskriptivní popis transformace. To znamená, že definujeme, co se má převádět, ale není zde již definováno jak. Nejčastěji se XML data transformují zpět do XML nebo do HTML.

Saxon Jedná se o XSLT a XQuery procesor [9, 10] napsaný v programovacím jazyce Java. Základní verze Saxonu je opensource. Existuje i komerční verze, která obsahuje další rozšiřující nástroje. Saxon tedy umí pomocí XSLT transformovat XML soubory, zpracovat data pomocí XQuery dotazovacího jazyku. Samozřejmostí je také podpora XPath, které zajišťuje navigaci v XML dokumentu.

3 XQuery

XQuery 1.0 [12, 13] je velmi komplexní dotazovací jazyk operující nad XML daty. Základní výrazy přebírá XQuery z dříve zmíněného XPath. Vazba XQuery k XML dokumentu je považována za stejnou jako je SQL pro relační databázi. Právě z SQL si vzal důležité konstrukty jako jsou FLWOR. XQuery jako takové se inspirovalo i v jiných jazycích. XQuery je založeno na **deklarativním programování** [11]. To znamená, že se určuje, co se má vykonat, ale není zde již definováno, jakým způsobem.

3.1 Užití XQuery

Samotné XML se hojně používá k ukládání strukturovaných dat. Je tedy samozřejmé, že dříve, či později, budeme potřebovat uložená data prohledávat. K tomuto účelu slouží dotazovací jazyk XQuery, který byl navržen konsorciem W3C. V současné době se používá verze 1.0.

3.2 Vlastnosti XQuery

Dva velmi důležité aspekty návrhu XQuery jsou, že se jedná o funkcionální jazyk založený na typech. Tyto aspekty hrají důležitou roli v XQuery formální sémantice. Cílem formální sémantiky je definování významu jednotlivých XQuery výrazů.

1. XQuery je **funkcionální jazyk**, využívající dříve zmíněné deklarativní programovací principy. Je postaven na výrazech. Každý konstrukt v jazyce XQuery, mimo prolog, se skládá z výrazů a tyto výrazy se mohou libovolně skládat mezi sebou. Výsledek jednoho výrazu může být použit jako vstup pro jakýkoli jiný výraz. Platí zde ovšem podmínka, že typ výsledku předchozího výrazu musí být kompatibilní se vstupem následujícího výrazu, se kterým je spojen. Dalším charakteristickým rysem funkcionálního jazyka je, že proměnné nesou hodnoty během zpracování. Takže hodnotu nemůžeme ovlivnit žádnými vedlejšími účinky. Proměnná začíná prefixem $, například $cena.

2. XQuery je jazyk **založený na typech**. Typy mohou být definovány přímo v XQuery výrazu nebo mohou být importovány z XML Schéma. XQuery může provádět operace na základě těchto typů. Navíc XQuery podporuje statickou analýzu. Tato analýza odvozuje výstupní typ výrazu podle svých vstupů. Díky tomu také zajišťuje brzkou detekci chyb mezi nekompatibilními typy.

3.3 Výhody XQuery

Jak již bylo dříve zmíněno, XQuery je jazyk pokrývající obrovskou množinu operací nad XML daty. Mezi nejvyužívanější patří:

- FLWOR dotazy inspirované SQL jazykem.
- Výstup jednoho dotazu můžeme použít jako vstup navazujícího.

- Provázání s XML Schema. Podle připojeného schématu definujeme datové typy v XQuery.
- Volnost, jeden dotaz můžeme napsat více způsoby, i když vykonává naprosto to samé.

3.4 Rozdíl oproti XPath

XPath je podmnožina XQuery, která neobsahuje tak uživatelsky orientovanou syntaxi jako XPath. Jinými slovy, XQuery je obohaceno o velmi užitečné funkce, které se v XPath nevyskytují, jedná se například o práci s kvantifikátory, FLWOR příkazy, uživatelsky definované funkce atd. Existují ale i společné vlastnosti, XQuery používá k navigaci ve stromové struktuře dokumentu stejnou posloupnost přechodů jako u XPath.

3.5 XQuery Update Facility

Jedná se o rozšíření jazyka XQuery, které umožňuje aktualizace stávajících XML dokumentů.
Funkce XQuery Update Facility [29]:

- Vložení uzlů - Možnost určit pozici (after, before), případně pro děti uzlu (as first, as last)
- Smazání uzlů - Lze využít podmínky pro smazání. Například můžeme smazat všechny záznamy o produktech, které nejsou na skladě (počet = 0).
- Kopírování uzlů.
- Modifikování uzlů a jeho hodnot.

Následující příklad vykonává tuto funkcionalitu. Zjistí jestli element **e** má atribut **last-updated**. Pokud ano, tak modifikuj hodnotu tohoto atributu na aktuální datum. Pokud takový atribut element **e** neobsahuje, tak vytvoř tento atribut a vlož do něj aktuální datum.

```
if (fn:exists($e/@last-updated))
then replace value of node
        $e/last-updated with fn:current-date()
else insert node
        attribute last-updated {fn:current-date()} into $e
```

Výpis 4: XQuery Update

3.6 FLWOR konstrukce

Jedná se o speciální příkazy, které jsou velmi podobné těm, které známe z SQL.

3.6.1 Jednotlivé části FLWOR

Každé písmeno ze zkratky FLWOR označuje jeden konstrukt.

- **FOR** – Zpracujeme vstupní sekvenci a pro každý prvek v této sekvenci vrátíme určitý výsledek k dalšímu zpracování
- **LET** – Deklarování a přiřazení proměnné pro každý prvek posloupnosti
- **WHERE** – Podmínka pro prvky v posloupnosti
- **ORDER BY** – Seřazení prvků
- **RETURN** – Specifikování výstupu pro výsledné prvky

FLWOR gramatické pravidlo

Podle XQuery gramatiky se rozšiřuje neterminál FLWORExpr následujícím pravidlem.

```
FLWORExpr ::= (ForClause|LetClause)+ WhereClause? OrderByClause? "return" ExprSingle
```

Výpis 5: FLWOR gramatické pravidlo

Z tohoto gramatického přepisu můžeme určit, že FLWOR konstrukce začíná buď konstruktem FOR nebo LET. Tyto konstrukty se mohou vyskytovat jakkoli neomezeně za sebou. Následuje nepovinný konstrukt WHERE, který určuje podmínku pro celou konstrukci. Dalším nepovinným konstruktem je ORDER BY, který slouží k seřazení výsledku. Poslední částí je RETURN, který definuje, jakým způsobem budou vypsány výsledné prvky.

```
ForClause ::= "for" "$" VarName TypeDeclaration? PositionalVar? "in" ExprSingle ("," "$" VarName
    TypeDeclaration? PositionalVar? "in" ExprSingle)*
LetClause ::= "let" "$" VarName TypeDeclaration? ":=" ExprSingle ("," "$" VarName
    TypeDeclaration? ":=" ExprSingle)*
TypeDeclaration ::= "as" SequenceType
PositionalVar ::= "at" "$" VarName
WhereClause ::= "where" ExprSingle
OrderByClause ::= (("order" "by") | ("stable" "order" "by")) OrderSpecList
OrderSpecList ::= OrderSpec ("," OrderSpec)*
OrderSpec ::= ExprSingle OrderModifier
OrderModifier ::= ("ascending" | "descending")? ("empty" ("greatest" | "least"))? ("collation"
    URILiteral)?
```

Výpis 6: Zbývající gramatická pravidla pro FLWOR konstrukce

Zbývající pravidla týkající se FLWOR konstruktů jsou zobrazeny ve výpisu výše. Jedná se o výrazy

Symboly opakovatelnosti *, + a ? jsou popsány v kapitole 5.4 zabývající se JavaCC a JJTree.

3.6.2 Ukázka FLWOR konstrukce

Následující příklad nám vypíše seznam oddělení s alespoň deseti zaměstnanci, seřazených sestupně podle průměrné výplaty [14].
Tato FLWOR konstrukce využívá všechny konstrukty. Nejprve pomocí FORu projdeme všechny čísla oddělení. Během procházení zjišťujeme informace o zaměstnancích aktuálně zpracovávaného oddělení. Pomocí podmínkového konstruktu vybereme oddělení, kde je alespoň deset zaměstnanců. Setřídíme sestupně a vypíšeme podle potřeby do požadovaného XML výstupu.

```
for $d in document("oddeleni.xml")//cisloOddeleni
let $e := document("zamestnanci.xml")//zamestnanec[cisloOddeleni = $d]
where count($e) >= 10
order by avg($e/vyplata) descending
return
  <oddeleni>
    { $d,
      <pocetZamestnancu>{count($e)}</pocetZamestnancu>,
      <vyplata>{avg($e/vyplata)}</vyplata>
    }
  </oddeleni>
```

Výpis 7: Ukázka FLWOR dotazu

4 Normalizace XQuery

Jak bylo již dříve zmíněno, normalizace jazyka XQuery je hlavní náplní této práce. V této kapitole si popíšeme důvody normalizace, jakým způsobem jsou normalizační pravidla zadána a nakonec aplikujeme jejich užití na zadané výrazy. Pro lepší pochopení jsou zde uvedeny také obrázky, které znázorňují transformace způsobené normalizačními pravidly.

4.1 Důvody normalizace

Pomocí normalizace [15] převedeme vstupní výraz využívající XQuery gramatiku na výraz využívající gramatiku XQuery Core, která již neobsahuje redundantní gramatická pravidla. Zeštíhlenou Core gramatiku je následně lehčí definovat, implementovat a optimalizovat.

4.2 Seznámení s normalizačními pravidly

Pro lepší pochopení normalizačních pravidel se v následující části obeznámíme s jeho obecným zápisem a jednoduchým příkladem na přepis.
Normalizační pravidlo je obecně zadáno takto:

$$[Expr]_{Expr} == CoreExpr$$

Expr v hranatých závorkách označuje původní XQuery výraz, který je normalizován na výraz CoreExpr. Dolní index $_{Expr}$ vyjadřuje normalizační pravidlo, které je aplikováno na tento výraz.
Některá pravidla se během normalizace nemění. Například přepis pro literály nebo proměnné je vždy totožný.

$$[IntegerLiteral]_{Expr} == IntegerLiteral$$

Výsledné pravidlo na pravé straně může také obsahovat další zápisy v hranatých závorkách. Ty určují další normalizace. Například rozsahový operátor **to**. Zápis **(1 to 5)** je identický se zápisem **(1,2,3,4,5)**. Tento výraz je normalizován pomocí tohoto pravidla:

$$[Expr_1\ to\ Expr_2]_{Expr} == fs:to(([Expr_1]_{Expr}), ([Expr_2]_{Expr}))$$

Všimněte si, že operátor **to** je normalizován do funkce **fs:to**. Tato funkce má v XQuery gramatice následující zápis (náš příklad neobsahuje nepovinné části určující datové typy zadaných hodnot):

```
fs:to ($ firstval as xs:integer?, $lastval as xs:integer?) as xs:integer*
```

Výpis 8: Funkce fs:to

Následně jsou zpracovány pravidla na pravé straně $[Expr_1]_{Expr}$ a $[Expr_2]_{Expr}$.
V jednom z předchozích pravidel jsme si vysvětlili, že literály nejsou normalizací ovlivněny. Z toho vyplývá, že úplný přepis normalizovaného pravidla z předchozího příkladu, doplněného o hodnoty **(1 to 5)**, vypadá takto:

$$[1\ to\ 5]_{Expr} == fs:to(([1]_{Expr}),([5]_{Expr})) == fs:to((1),(5))$$

4.3 FLWOR normalizace

Tato práce je zaměřena především na FLWOR normalizaci. FLWOR konstrukty jsou normalizovány do vnořených Core FLWOR konstruktů s pouze jedním FOR nebo LET konstruktem. Toto omezení zajišťuje, že LET a FOR jsou svázány pouze s jednou proměnnou. Více o těchto pravidlech je popsáno v jednotlivých podkapitolách.

4.3.1 FOR normalizace

Nejdůležitější část normalizace konstruktu FOR spočívá hlavně v přepisu na nezávislé FOR konstrukty. Tímto přepisem docílíme, že jeden FOR konstrukt je vždy svázán pouze s jednou proměnnou. Nepovinné neterminály $OptTypeDeclaration$ a $OptPositionalVar$ jsou pro normalizaci nepodstatné, protože se během normalizace nemění.

$$
\begin{aligned}
&[\ for\ \$VarName_1\ OptTypeDeclaration_1\ OptPositionalVar_1\ \text{in}\ Expr_1,\\
&\dots,\\
&\$VarName_n\ OptTypeDeclaration_n\ OptPositionalVar_n\ \text{in}\ Expr_n\\
&FormalReturnClause\ ]_{Expr}\\
&==\\
&for\ \$VarName_1\ OptTypeDeclaration_1\ OptPositionalVar_1\ \text{in}\ [\ Expr_1\ ]_{Expr}\ return\\
&\dots\\
&for\ \$VarName_n\ OptTypeDeclaration_n\ OptPositionalVar_n\ \text{in}\ [\ Expr_n\ ]_{Expr}\ return\\
&[\ FormalReturnClause\ ]_{Expr}
\end{aligned}
$$

Obrázek 2: Normalizační pravidlo FOR

Využití pravidla Nyní si předvedeme velmi jednoduchý příklad aplikování FOR normalizace.

```
for $i in (1, 2),
    $j in (4 to 8)
return
    element pair { ($i,$j) }
```

Výpis 9: Původní FLWOR konstrukce obsahující FOR konstrukty

Normalizace zajistí, že jeden FOR konstrukt je svázán pouze s jednou proměnnou. Následující proměnné jsou převedeny do následujících FOR konstruktů, oddělených klauzulí return. Nesmíme také zapomenout na rozsahový operátor **to**, který je dále normalizován. Normalizována konstrukce vypadá takto:

```
for $i in (1, 2) return
    for $j in ( fs:to ((4) ,(8) )) return
        element pair { ($i ,$j ) }
```

Výpis 10: Normalizována FLWOR konstrukce obsahující FOR konstrukty

4.3.2 LET normalizace

Normalizační pravidlo pro konstrukt LET (zobrazeno na obrázku 3) je velmi podobné, jako pravidlo pro FOR. Podle gramatiky jsou přípustné různé kombinace FOR a LET konstrukcí. Kombinace těchto konstruktů a jejich normalizace je zobrazena ve výpisu číslo 11.

$$
\begin{aligned}
&[\,let\ \$\,VarName_1\ OptTypeDeclaration_1 := Expr_1\,,\\
&\cdots,\\
&\$\,VarName_n\ OptTypeDeclaration_n := Expr_n\\
&FormalReturnClause\,]_{Expr}\\
&==\\
&let\ \$\,VarName_1\ OptTypeDeclaration_1 := [\,Expr_1\,]_{Expr}\ return\\
&\cdots\\
&for\ \$\,VarName_n\ OptTypeDeclaration_n := [\,Expr_n\,]_{Expr}\ return\\
&[\,FormalReturnClause\,]_{Expr}
\end{aligned}
$$

Obrázek 3: Normalizační pravidlo LET

4.3.3 WHERE normalizace

Zajímavý je přepis týkající se podmínkového konstruktu WHERE. Ten je přepsán na syntaxi **If-Then-Else**. Výraz definovaný za WHERE je přepsán do uzávorkované podmínky If. Klauzule následující za konstruktem RETURN (značena také $FormalReturnClause$) je zapsána za Then. Klauzule Else je u tohoto přepisu vždy prázdná, protože v konstruktu WHERE nedefinujeme, co se má provést v případě, že výraz nesplní zadanou podmínku. Samozřejmě, že u pravidla **If-Then-Else** se může vyskytnout výraz za Else.

Po získání **If-Then-Else** je ještě tato syntaxe zpracována svým vlastním normalizačním pravidlem.

$$
\begin{aligned}
&[\,\mathit{where}\ \mathit{Expr}_1\ \mathit{FormalReturnClause}\,]_{Expr}\\
&==\\
&\mathit{if}\,([\,\mathit{Expr}_1\,]_{Expr})\ \mathit{then}\ [\,\mathit{FormalReturnClause}\,]_{Expr}\ \mathit{else}\,()
\end{aligned}
$$

Obrázek 4: Normalizační pravidlo WHERE

4.3.4 ORDER BY normalizace

Třídící konstrukt ORDER BY je normalizován ve dvou krocích. Nejprve se převede do seznamu $OrderSpecList$, který obsahuje elementy, podle kterých se má třídit a také jakým způsobem (sestupně nebo vzestupně). Následuje klauzule return, která definuje výstupní formát. Druhá část normalizace spočívá v převedení dříve zmíněného seznamu $OrderSpecList$ na uskupení LET konstruktů, které vykonají třídění pomocí algoritmu bublle sort [16] .
Tato normalizace není implementována v této práci, protože tímto zdlouhavým přepisem ztrácíme deklarativní formát výsledku [11].

$$
\begin{aligned}
&[\,\mathit{stable\ ?\ order\ by\ OrderSpecList\ FormalReturnClause}\,]_{Expr}\\
&==\\
&[\,\mathit{OrderSpecList}\,]_{OrderSpecList}\ \mathit{return}\ [\,\mathit{FormalReturnClause}\,]_{Expr}
\end{aligned}
$$

$$
\begin{aligned}
&[\,\mathit{OrderSpecList}\,]_{OrderSpecList}\\
&==\\
&\mathit{LetClause}\ldots\mathit{LetClause}
\end{aligned}
$$

Obrázek 5: Zjednodušené normalizační pravidlo ORDER BY

4.3.5 RETURN normalizace

Poslední konstrukt je na rozdíl od všech ostatních nezávislý. Díky tomu se normalizace nevykonává vůbec. Klíčové slovo **return** je přidáno již při předchozích normalizačních přepisech.

$$[return\ Expr]_{Expr} == [Expr]_{Expr}$$

4.3.6 Příklad FLWOR normalizace

Jak bylo zmíněno dříve, konstrukty FOR a LET můžeme jakkoliv kombinovat. Jejich normalizační pravidla jsou velmi obdobná. Jedná se pouze o rozložení složených konstruktů na jednotlivé menší části. Tento rozklad nám zaručuje bezproblémovou práci s libovolnou

kombinací těchto konstruktů. Následuje ukázka normalizace FLWOR konstrukce, která obsahuje právě kombinace konstruktů FOR a LET. V tomto příkladě můžeme také vidět normalizaci konstruktu WHERE.

```
for $i in (1, 4, 9), $j in 1 to 5
let $k := $i + $j
where $k > 25
return ($k − $j)
```

Výpis 11: Původní FLWOR konstrukce

```
for $i in (1, 4, 9) return
    for $j in 1 to 5 return
        let $k := $i + $j return
            if (fn:boolean ($k >25 ))
               then $k − $j
               else ()
```

Výpis 12: Normalizovaná FLWOR konstrukce

Funkce **fn:boolean** je přidána díky normalizaci kontrukce If-Then-Else 4.4.3, která vznikla přepisem normalizačního pravidla pro konstrukt Where.

4.4 Ostatní normalizace

Normalizační pravidla jsou definována také pro další gramatická pravidla, než jen pro FLWOR konstrukce. V této podkapitole si představíme ta nejdůležitější, která jsou implementována v této práci.

4.4.1 Normalizace cest

Zadání cesty Nejprve se zaměříme na normalizaci lomítka. Od této normalizace se odvíjejí i další dvě pravidla zobrazena na obrázku 6. Cestu zadáváme posloupností přechodů mezi jednotlivými sadami uzlů, oddělenými lomítky. Jedná se o přepis gramatického pravidla **RelativePathExpr**.
Můžete si všimnout, že všechna pravidla využívají funkci **fn:root**, která vrací kořenový element (nejvyššího předka). Rozdíl mezi druhým a třetím pravidlem je ve výběru. Pomocí dvou lomítek vybíráme všechny potomky daného uzlu. S jedním lomítkem vybíráme pouze své děti. U posledního pravidla stojí za povšimnutí rozšíření o osu **descendant-or-self ::node()**, která zajišťuje nejen výběr v samotném uzlu, ale také ve všech jeho potomcích.

$$
\begin{aligned}
&[/]_{Expr} \\
&== \\
&[((fn:root(self::node()))treat\ as\ document-node())]_{Expr}
\end{aligned}
$$

$$
\begin{aligned}
&[/\ RelativePathExpr\ ]_{Expr} \\
&== \\
&[((fn:root(self::node()))treat\ as\ document-node())/\ RelativePathExpr\ ]_{Expr}
\end{aligned}
$$

$$
\begin{aligned}
&[//\ RelativePathExpr\ ]_{Expr} \\
&== \\
&[((fn:root(self::node()))treat\ as\ document-node())\ \ / \\
&descendant-or-self::node()/RelativePathExpr\ ]_{Expr}
\end{aligned}
$$

Obrázek 6: Normalizační pravidlo RelativePathExpr

ContextItem Tento výraz vyjadřuje spojitost s aktuálním uzlem v XML dokumentu. Normalizace je prováděna pomocí proměnné **$fs:dot**.

$$[.]_{Expr} == \$fs:dot$$

4.4.2 Normalizace os

Osy slouží k popsání struktury XML dokumentu vzhledem k aktuálnímu uzlu. Tímto způsobem můžeme definovat potomky, předchůdce, sousedy a další elementy v XML dokumentu, jako jsou jmenné prostory či atributy. Obrázek všech os je zobrazen v kapitole o XPath 1. Některé osy lze zapsat pomocí kombinace jiných výrazů. Jedná se o sousedící elementy, které jsou rozděleny na dva druhy a to předcházející a následující. Ostatní osy se nenormalizují.

Dopředné osy Osa **following-sibling** vybírá následující sourozence. Její přepis využívá konstrukt **let**. Nejprve zjistíme rodiče pomocí osy $parent :: node()$ a poté vybereme následující sourozence $child :: NodeTest[. >> \$e]$.
Osa **following** využívá předchozí pravidlo a rozšiřuje ho o všechny předchůdce a potomky následujících sourozeneckých elementů.
$>>$ patří mezi operátory určené k porovnávání uzlů. Vyhodnotí výraz jako pravdu, jestliže za uzlem na levé straně následuje uzel definovaný na straně pravé. V jiném případě je výraz vyhodnocen jako nepravda.

$$
\begin{aligned}
&[\ following-sibling::NodeTest\ ]_{Axis}\\
&==\\
&[\ let\ \$e:=.\ return\ \$e/\ parent::node()/child::NodeTest[\ .\gg\ \$e\ ]\ ]_{Expr}
\end{aligned}
$$

$$
\begin{aligned}
&[\ following::NodeTest\ ]_{Axis}\\
&==\\
&[\ ancestor-or-self::node()/\ following-sibling::node()/descendant-or-self::NodeTest\ ]_{Expr}
\end{aligned}
$$

Obrázek 7: Normalizační pravidlo following a following-sibling

Zpětné osy Podobně jako dopředné osy following a following-sibling fungují zpětné osy **preceding** a **preceding-sibling** s tím rozdílem, že vybírají předchozí sourozence, eventuálně i jejich potomky a předky. Protože se jedná o zpětné osy, tak musíme použít opačný operátor $<<$, než který jsme použili v předchozím případě.

$$
\begin{aligned}
&[\ preceding-sibling::NodeTest\ ]_{Axis}\\
&==\\
&[\ let\ \$e:=.\ return\ \$e/\ parent::node()/child::NodeTest[\ .\ll\ \$e\ ]\ ]_{Expr}
\end{aligned}
$$

$$
\begin{aligned}
&[\ preceding::NodeTest\ ]_{Axis}\\
&==\\
&[\ ancestor-or-self::node()/\ following-sibling::node()/descendant-or-self::NodeTest\ ]_{Expr}
\end{aligned}
$$

Obrázek 8: Normalizační pravidlo preceding a preceding-sibling

Zkrácené zápisy Při psaní XQuery výrazu se často používají zkratky. Normalizace přepíše tyto zkratky do jejich nezkrácené podoby.

$$
\begin{aligned}
&[..]_{Axis}\\
&==\\
&parent::node()
\end{aligned}
$$

$$
\begin{aligned}
&[@\ NodeTest\ ]_{Axis}\\
&==\\
&attribute::NodeTest
\end{aligned}
$$

Obrázek 9: Zkrácené zápisy

Zápis **..** se přepisuje na rodičovskou osu. Zavináč se používá pro definování atributu.

4.4.3 If-Then-Else

Tento přepis je velmi triviální. Každý z výrazů se zvlášť dále normalizuje. Navíc je přidána funkce $fn : boolean$, která zajistí vyhodnocení podmínky, buď jako pravdu nebo nepravdu.

$$
\begin{aligned}
&[\mathit{if}\ (\mathit{Expr}_1)\ \ \mathit{then}\ \ \mathit{Expr}_2\ \ \mathit{else}\ \ \mathit{Expr}_3]_{Expr} \\
&== \\
&\mathit{if}\ (\mathit{fn}:\mathit{boolean}([\mathit{Expr}_1]_{Expr}))\ \ \mathit{then}[\mathit{Expr}_2]_{Expr}\ \ \mathit{else}[\mathit{Expr}_3]_{Expr}
\end{aligned}
$$

Obrázek 10: Normalizační pravidlo If-Then-Else

5 Lexikální a syntaktická analýza

V této kapitole si popíšeme prostředky, které nám pomohou jednoduše zpracovat určitou posloupnost a převést ji do potřebné výstupní formy.

5.1 Využití parser generátoru

Abychom mohli implementovat normalizaci jazyka XQuery, tak nejprve potřebujeme převést tento výraz do stromové struktury. K tomuto procesu se skvěle hodí parser generátor [17, 18, 24].
Generátor parserů je nástroj, který vytváří parser z formálně popsaného vstupu, nejčastěji je popsán formální gramatikou. Typický parser generátor generuje určitý zdrojový kód pro každé gramatické pravidlo popsané formální gramatikou. Abychom tuto definici lépe pochopili je nutno vysvětlit pojmy jako parser, lexikální a syntaktická analýza.

5.1.1 Lexikální analýza

Lexikální analýza [19] je proces, který nám rozdělí vstupní posloupnost znaků na lexémy, ty jsou dále reprezentovány jako tokeny. Lexikální analýza se řídí lexikálními pravidly, které jsou většinou definovaná pomocí regulárních výrazů.
Tento proces se musí vykonat před syntaktickou analýzou, protože výstup lexikální analýzy je vstupem syntaktické analýzy.

Hlavní funkce Hlavní funkce lexikálního analyzátoru:

- Rozdělení na tokeny. Token se skládá ze dvou částí. Jedná se o typ (identifikátor, číslo nebo operátor, atd.) a lexém (řetězec, hodnota, kód operátoru, atd.).
- Odstranění komentářů a bílých znaků ze zdrojového programu.
- Normalizace symbolů - Například určení case sensitive (citlivosti na velikost písmen).

Například lexikální analýza pro vstup **sum=4*3+2;** je v následující tabulce 1.

5.1.2 Syntaktická analýza

Syntaktická analýza, neboli parsování, je proces, který zpracovává vstupní text (tokeny) a transformuje jej do určité datové struktury, například strom, podle předem definované formální gramatiky. Důležitou vlastností vygenerované struktury je zachování hierarchie vstupních dat. Ukázka gramatiky ve výpisu 13 a vygenerovaného syntaktického stromu **sum=4*3+2;** na obrázku 11.

lexém	typ tokenu
sum	Identifikátor
=	Rovnítko
4	Číslo
*	Krát
3	Číslo
+	Plus
2	Číslo
;	Oddělovač

Tabulka 1: Lexikální analýza pro vstup sum=4*3+2;

```
S −> id=A;
A −> M ( ( "+" | "−" ) M )∗
M −> U ( ( "∗" | "/" | "%" ) U )∗
U −> "(" A ")" | L
L −> num

sum = 4 ∗ 3 + 2;
S −> id = A; −> id = M + M; −> id = U ∗ U + U; −> id = num ∗ num + num;
```

Výpis 13: Ukázka gramatiky generující syntaktický strom

5.1.3 Parser

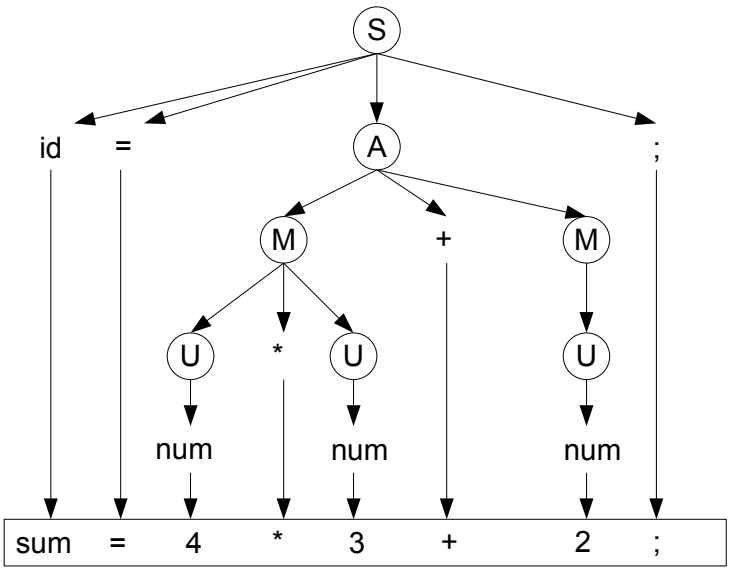
Parser je program, který vykonává parsování, neboli syntaktickou analýzu. Jedná se o proces, při kterém převádíme zadanou posloupnost prvků do určité datové struktury, která zachovává hierarchii vstupních dat. V našem případě se jedná o strom. Vytvoření stromu se provádí podle předem definovaných gramatických pravidel. Uskupení těchto pravidel se nazývá formální gramatikou. Navíc, některé parsery dokáží kontrolovat posloupnost prvků s ohledem na XML schéma.

5.1.4 Další druhy parser generátorů

Samozřejmě, že existuje velká řada parser generátorů. Mezi prvními a velmi rozšířenými byly lex a yacc [18, 19]. **Lex** slouží ke generování lexikálního analyzátoru. Syntaktická část je prováděna programem **yacc**, který generuje parser. Využívá k tomu výstup z jazyka lex. Oba programy generují zdrojový kód v programovacím jazyce **C**.

5.2 JavaCC

JavaCC [20, 21] je nejpopulárnější parser generátor pro použití v Java aplikacích. Parser generátor je nástroj, který čte gramatiku zadanou pomocí gramatických pravidel a pře-



Obrázek 11: Syntaktický strom

vádí ji na zdrojový kód v jazyce Java. Přeložený zdrojový kód dokáže rozpoznat, zda zadaný vstup odpovídá příslušné gramatice. Kromě samotného parser generátoru, JavaCC nabízí i další standardní funkce související s generováním parseru, jako je JJTree, nástroje pro ladění a podobné.

JavaCC je napsán v programovacím jazyce Java. Dokáže pracovat s Java VM od verze 1.2. JavaCC byl testován na bezpočtu různých platformách bez jakýchkoli problémů. Program je možné stáhnout ze stránek [20]. Zde můžeme najít jednoduché příklady již předvytvořených gramatik.

JavaCC generuje parsery shora-dolů, jedná se o takzvaný rekurzivní sestup.

5.3 JJTree

Jedná se o preprocesor pro JavaCC. JJTree [20, 22] vloží na specifikovaná místa akce, které se využívají pro sestavení parsovacího stromu. Výstup z JJTree musí být následně zkompilován přes JavaCC. Defaultně JJTree generuje kód pro každý neterminál, který se v gramatice objevuje. Toto chování lze samozřejmě modifikovat. Buď se pro některé neterminály nevytvářejí uzly vůbec, nebo se vytvoří uzly pouze pro část rozšíření neterminálu.

JJTree působí v jednom ze dvou módů simple a multi. V simple módu každý uzel stromu je typu SimpleNode. V Multi módu je typ uzlu určen podle jména uzlu. JJTree

vygeneruje základní funkcionalitu pro každý uzel. Samozřejmě, že tuto funkcionalitu můžeme modifikovat podle svých potřeb.

Přestože JavaCC parsuje shora-dolů, tak JJTree přesně naopak konstruuje parsovací strom zdola-nahoru. K tomu potřebuje zásobník, do kterého vkládá uzly ihned po jejich vytvoření. Když najde rodiče těchto uzlů, tak je vyjme ze zásobníku a přidá je pod rodiče. Nakonec vloží do zásobníku rodičovský uzel.

Ukázka JJTree souboru Zde je seznam nejdůležitějších částí.

- Nastavení - V této úvodní části můžeme mimo jiné definovat tyto hodnoty:
  - Název balíčku (package), ve kterém budou vygenerované soubory uloženy.
  - Prefix pro jednotlivé uzly.
  - Možnost využití návrhového vzoru Visitor.
  - Určení módu (Simple nebo Multi).
- Java zdrojový kód. Ačkoli je JJTree a JavaCC navržen, aby ušetřil práci s psaním zdrojového kódu, tak určitou část kódu zde můžeme také naimplementovat. Jedná se například o nastavení výpisu v případě neúspěšného vygenerování parseru.
- Definování tokenů a bílých znaků.
- Gramatická pravidla. Jeden neterminál odpovídá jedné metodě. Zápis pro FLWOR konstrukci vypadá takto:

```
void FLWORExpr() :
{}
{
  ((ForClause() | LetClause()))+  [WhereClause()] [OrderByClause()] "return" ExprSingle()
}
```

Výpis 14: Ukázka definování gramatického pravidla v JJTree

5.4 Výhody JavaCC a JJTree

Největší výhody využití JavaCC a JJTree jsou následující:

- Použití připraveného JJTree souboru, který obsahuje implementaci realizující zpracování XQuery výrazu.
- JavaCC zpracovává parsery shora-dolů, jedná se o takzvaný rekurzivní sestup. Výhodou toho zpracování je využití obecnějších gramatik a snadnější ladění.
- JJDoc je program, který je schopen ze souboru s gramatikou vygenerovat zápis gramatiky v běžném zápisu BNF. JJDoc je také součástí instalace JavaCC.
- JavaCC nemá problém zpracovat vstup v Unicode formátu.

- Defaultně JavaCC vytváří LL(1) gramatiky [23]. Nicméně můžeme vytvářet i gramatiky, které jsou LL(k). Část LL(k) gramatiky je definována pomocí příkazu LOOKAHEAD(k). Zbylé části jsou zpracovány jako LL(1). Pokud bychom zpracovávali celou gramatiku jako LL(k), mohlo by dojít ke značnému zpomalení zpracování vstupního souboru. Tím rozčleněním můžeme docílit jednoduchého zápisu gramatiky bez výrazného zpomalení parseru.
- JavaCC poskytuje srozumitelné chybové hlášení pro kompilaci. Daleko obtížnější je dohledat chybu, která se projeví až ve vygenerovaném zdrojovém kódu.
- Pokud potřebujeme důkladný rozbor kroků, které generátor provádí, můžeme zapnout ladění. JavaCC nám poté přesně hlásí, co a jak se provádí.
- JavaCC je naprogramován celý v programovacím jazyce Java. Lze ho tedy používat na mnoha různých platformách, stejně jako jím vygenerované parsery.
- Gramatiku v JavaCC lze zapisovat ve tvaru EBNF (Niklaus Wirth's Extended Backus-Naur Form) [30]. EBNF využívá symboly, které definují míru opakovatelnosti. EBNF umožňuje zápis ve formátu y(x)*. Tento zápis je lépe čitelný, než zápis v BNF. Ten by v našem případě vypadal například A ::= Ax|y. Symboly vyjadřující opakovatelnost v EBNF.
  - **?** - Symbol nebo skupina symbolů v závorkách se opakuje nejvýše jednou, nemusí se však vyskytovat vůbec.
  - * - Opakování není nijak omezeno. Symbol se může vyskytnout kolikrát chce, ale může být také přeskočen.
  - **+** - Symbol se objeví jednou nebo vícekrát

5.5 Rekurzivní sestup

Jedním z nejpoužívanějších způsobů parsování je metoda rekurzivního sestupu [24]. V této metodě se pomocí rekurzivních volání zpracovává syntaktický strom. Tento strom odpovídá příslušné gramatice. Jestliže se v gramatice objevuje na daném místě neterminální symbol, tak v programu mu odpovídá (rekurzivní) volání reprezentující příslušný neterminál. Navíc ve stromu je tento neterminál zobrazen jako uzel. Tato rekurzivní volání končí u terminálních symbolů (lexémů jazyka). Tyto terminální symboly odpovídají listům ve stromu.

6 Implementace

Tato kapitola obsahuje různorodé informace týkající se implementace.

6.1 Implementační prostředky

Jedná se o soubor prostředků, kterými byla tato diplomová práce naimplementována.

6.1.1 Java

Java [26, 27] je programovací jazyk pocházející od firmy Sun Microsystems. Jedná se o objektově orientovaný jazyk vycházející z C++.
Implementaci normalizačních pravidel v tomto programovacím jazyce jsem si vybral hlavně díky těmto vlastnostem:

- Java je jazyk velice jednoduchý, přehledný a srozumitelný.
- Značným ulehčením pro programátory je obsáhlost standardně dodávaných knihoven [28].
- Velmi dobrá práce s XML soubory.
- Přidělování a uvolňování paměti je zde obstaráno automaticky (pomocí garbage collectoru).
- Nezávislost na architektuře. Vytvořená aplikace běží na libovolném operačním systému nebo libovolné architektuře.

Velkou výhodou Javy je také její hardwarová nezávislost, neboť je překládaná do speciálního mezikódu, který je na konkrétním počítači nebo zařízení (PC, mobilní telefon apod.) interpretován, případně za běhu překládán do nativního kódu.

6.1.2 JavaCC, JJTree ruční překlad

Jak již bylo zmíněno v kapitole o JavaCC a JJTree (5.2), jedná se o programy, které nám pomáhají vytvořit parser podle zadané gramatiky. K získání zdrojového kódu, který také generuje stromovou strukturu definovanou v JJTree, jsou nutné dva kroky.

1. Vytvoření a následné zkompilování souboru *.jjt, který obsahuje gramatická pravidla a také předdefinované funkce, které ulehčují práci se stromem. Výstupem tohoto procesu je JavaCC soubor, který je vstupem následujícího kroku. Kromě tohoto souboru jsou také vytvořeny java třídy implementující dříve zmíněné stromové funkce.
2. Zkompilování JavaCC souboru (*.jj). Tento proces již vytvoří základní třídy parseru.

6.1.3 Eclipse plug-in

Předchozí postup je sice účinný, ale poměrně zdlouhavý. Osvědčila se nám instalace pluginu pro vývojové prostředí Eclipse [25]. Po instalaci stačí vytvořit nový JavaCC projekt s podporou JJTree. Zobrazí se ukázkový příklad implementace JJTree, který si každý upraví podle svých potřeb. Výhody práce s pluginem jsou:

- Všechny zdrojové kódy jsou zpracovány jedním vývojovým prostředím.
- Odpadá nutnost kompilovat JJTree a JavaCC sobory přes příkazový řádek. Při každém uložení je provedena dvojitá kompilace. Nejprve z JJTree souboru do JavaCC a poté z JavaCC přímo do java tříd.
- Předdefinované grafické prostředí pro práci s JJTree a JavaCC.

6.2 Struktura implementace

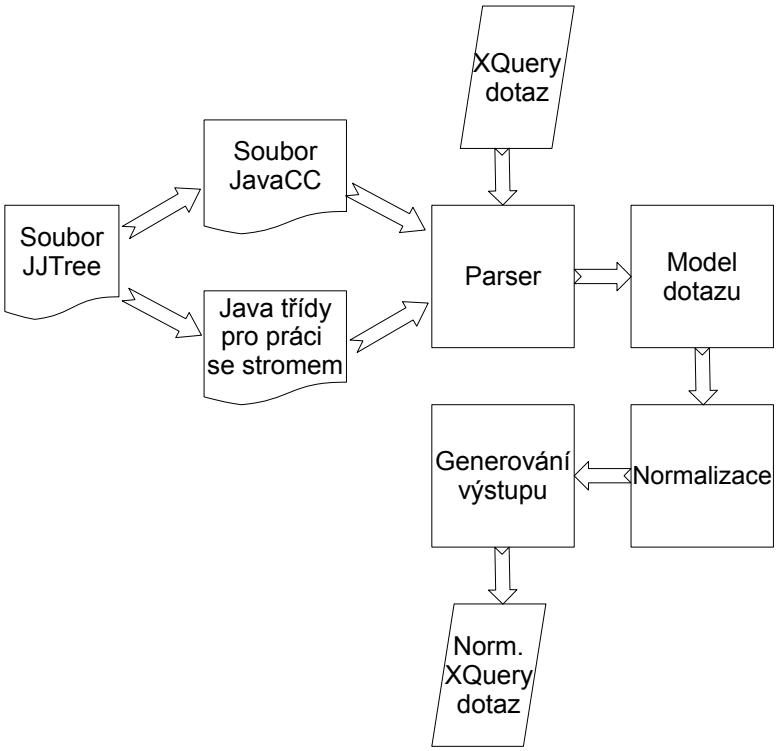
Již víme, že program byl naimplementován v programovacím jazyce Java s využitím generátorů JJTree a JavaCC. V této kapitole si zobrazíme celý průběh implementace. Detailněji se zaměříme na úpravy provedené v JJTree, seznam všech tříd s krátkým popisem a model XQuery dotazu převedeného do stromové struktury.

6.2.1 Implementační proces

V této části si znázorníme kroky, které vedly od využití JJTree souboru až po generování normalizovaného XQuery výstupu. Následující seznam a obrázek 12 zobrazuje všechny postupné kroky implementace.

- **Soubor JJTree** - Obsahuje implementaci XQuery parseru včetně základních metod pro práci se stromem.
- **Soubor JavaCC** - Vygenerovaný soubor, který obsahuje potřebné informace z JJTree pro generování parseru.
- **Java třídy pro práci se stromem** - Tyto třídy jsou vygenerované z původního JJTree souboru (například SimpleNode).
- **Parser** - Již funkční parser vygenerovaný z JJTree a JavaCC souborů.
- **XQuery dotaz** - Vstupní nenormalizovaný výraz.
- **Model dotazu** - Dotaz převedený do stromové struktury.
- **Normalizace** - Vykonání normalizačních pravidel na dotaz reprezentovaný stromem.
- **Generování výstupu** - Zpracování normalizovaného stromu a generování příslušného výstupu pro každý uzel.

- **Norm. XQuery dotaz** - Výsledný normalizovaný XQuery dotaz využívající XQuery Core gramatiku.



Obrázek 12: Implementační proces

6.2.2 Tvorba a upravení JJTree

Mezi základní kameny této práce patří JJTree soubor, díky kterému můžeme rozpoznat XQuery gramatiku. Na stránkách konsorcia W3C [3] je uložena implementace parseru (v JJTree formátu) rozpoznávající nejen XQuery ale také XPath gramatiku. Tento soubor jsem využil. Nicméně jsem ho musel modifikovat, jednak kvůli výpisu skrytých normalizačních pravidel, ale také kvůli doplnění o metody zajišťující práci s uzly.
Využitím tohoto souboru jsme si značně ušetřili práci s implementací parseru. Po zkompilování JJTree a JavaCC souborů do Java tříd již můžeme pracovat s funkčním parserem, který přijímá XQuery výraz a vytváří k němu syntaktický strom.

Skrytá pravidla Ve výpisu 15 je zobrazeno gramatické pravidlo pro FLWOR konstrukci. Pokud je pravidlo zadáno jako skryté, s **#void**, tak se nevytvoří uzel odpovídající průchodu přes toto pravidlo. Během normalizace potřebujeme vytvářet uzel odpovídající gramatickému pravidlu pro $FLWORExpr$. Proto zápis **#void** v našem JJTree souboru

chybí. Vypuštění **#void** je provedeno téměř u všech gramatický pravidel, aby bylo možné přesně rozpoznat, jaký model měl původní XQuery výraz.

```
void FLWORExpr() #void :
{}
{
  ((ForClause() | LetClause()))+  [WhereClause()] [OrderByClause()] "return" ExprSingle()
}
```

Výpis 15: Ukázka skrytého gramatického pravidla FLWORExpr v JJTree

6.2.3 Vygenerované třídy

Tyto třídy byly nejprve vygenerovány pomocí JJTree, případně JavaCC souboru a poté byly podle potřeby změněny. Funkcionalita jednotlivých tříd a rozhraní je popsána v následujícím seznamu.

- `XPath` - Jedná se o hlavní třídu generující parser. Obstarává následující úkony.
  - Spouštěcí metoda - `public static void main(String args[])` - Zde jsou vstupní argumenty předány k dalšímu zpracování. XQuery výraz je převeden do stromové struktury a následně je poslán k normalizaci. Pokud je před výrazem zadáno **„-o“** nebo **„-a“**, tak se vytvoří pomocný soubor, který obsahuje podrobnější informace o provedené normalizaci.
  - Lexikální analýza - Zpracování jednotlivých tokenů XQuery výrazu. Během analýzy se využívají objekty tříd XPathTokenManager a Token.
  - Syntaktická analýza - Podle definovaných gramatických pravidel je vytvořena struktura metod, kde každá metoda odpovídá určitému pravidlu. Například u metody FLWORExpr(), zjišťujeme jaký bude následující token. Musí to být buď LET nebo FOR. Jestliže je jeden z těchto tokenu nalezen, tak se vykoná metoda ForClause(), respektive LetClause().
- `XPathConstants` - Jedná se o rozhraní, ve kterém jsou definovány terminální symboly (+,-,(,) a tak dále). Jsou také zde zaznamenány lexikální stavy.
- `XPathTokenManager` - Třída, ve které jsou definovány metody starající se o zpracování tokenů (lexikální analýzu).
- `XPathTreeConstants` - Jedná se o podobné rozhraní jako XPathConstants s tím rozdílem, že obsahuje neterminály gramatických pravidel.
- `XPathVisitor` - Využití návrhového vzoru Visitor.
- `TokenMgrError` - Detekování chyby při zpracování tokenů.
- `Token` - Třída definující tok (stream) tokenů.

- `Node` - Rozhraní pro práci s uzly, ke třídě SimpleNode.
- `SimpleNode` - Třída, která implementuje předchozí rozhraní. Obsahuje důležité metody
  - Práce s rodičovským uzlem. Nastavení a získání rodičovského uzlu.
  - Práce s dětským uzlem. Přidání, nahrazení, smazání uzlu. Získání dětských uzlů a jejich počet.
  - Rekurzivní výpis všech potomků.
- `ParseException` - Třída starající se o vypisování chybových zpráv.
- `SimpleCharStream` - Implementace rozhraní CharStream, které obsahuje pouze znaky z ASCII tabulky.
- `JJTXPathState` - Třída obstarávající funkcionalitu zásobníku, který je využíván při tvorbě stromu.

Všechny tyto třídy a rozhraní jsou umístěny v balíčku **org.w3c.xqparser**.

Odebrání dětského uzlu ve stromu Ve třídě SimpleNode jsem musel doimplementovat metodu pro odstranění dětského uzlu. Tato metoda je popsána pomocí pseudokódu ve výpisu 1. Ukázka pole $children[]$ je znázorněna na obrázku 13.

```
1  int i - pozice dětského uzlu určeného ke smazání;
2  children[] - pole obsahující dětské uzly;
3  Vytvoření pomocného pole c[], které obsahuje o 1 prvek méně než pole children[];
4  if i = 0 then
5  |  Zkopíruj pole children[] mimo nultého prvku do pole c[];
6  else
7  |  for j = 0; Dokud j je menší než délka pole children -1; j++ do
8  |  |  if j je větší než i then
9  |  |  |  c[j] = children[j];
10 |  |  else
11 |  |  |  c[j] = children[j + 1];
12 |  |  end
13 |  end
14 end
15 Přepiš pole children[] pomocným polem c[];
```

Algoritmus 1: Odstranění dětského uzlu

6.2.4 Normalizační třídy

V této kapitole si projdeme dvě třídy, které se starají o normalizaci vstupních výrazů a přepisy na výstupní normalizovaný tvar.

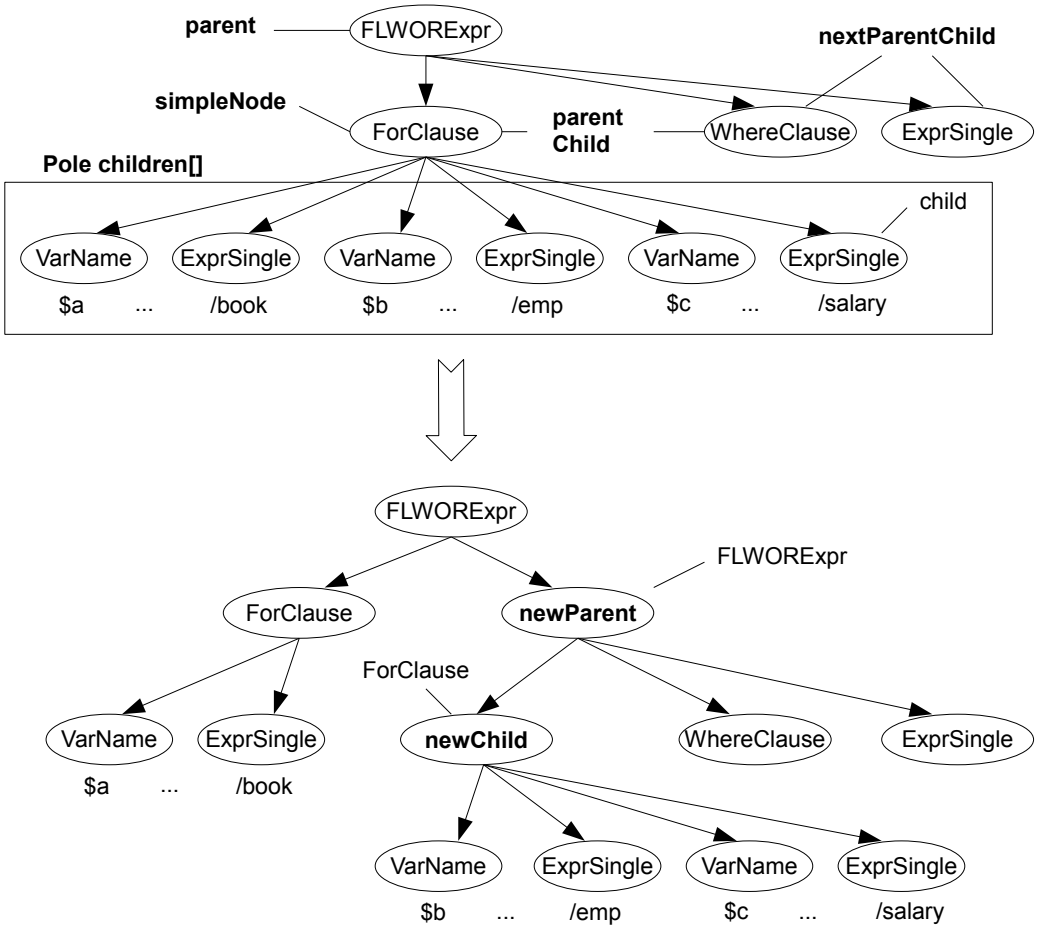
- `Main` - Hlavní metoda obsahuje následující metody.
  - Spuštění normalizace
  - Vypsání stromové struktury rekurzivním sestupem
  - Nastavení výstupního souboru
- `Normalization` - Třída implementující normalizaci.
  - Transformace vykonané na stromové struktuře.
  - Zpracování výstupního výrazu.

Tyto třídy jsou umístěny v balíčku **normalization**.

6.2.5 Model XQuery dotazu

Zadaný XQuery výraz je převeden do stromové struktury. Následující obrázek 13 ukazuje právě tuto stromovou reprezentaci pro normalizaci FOR nebo LET konstruktu. V tomto obrázku jsou definovány obecné uzly vzhledem k uzlu $simpleNode$. Na tyto obecné uzly se později odkazujeme během popisování algoritmu pomocí pseudokódu.

- $simpleNode$ - Aktuálně procházený uzel.
- $parent$ - Rodič aktuálního uzlu.
- $child$ - Dítě aktuálního uzlu uloženého v poli $children[]$.
- $parentChild$ - Všechny uzly na stejné úrovni jako $simpleNode$.
- $nextParentChild$ - Následující uzly na stejné úrovni jako $simpleNode$.
- $newParent$ - Nově vytvořený uzel pod uzlem $parent$.
- $newChild$ - Nově vytvořený uzel pod uzlem $newParent$.



Obrázek 13: Model výrazu

Ukázka prefixového průchodu Díky tomuto způsobu zpracování se uzly normalizují tak dlouho, dokud nejsou všichni potomci daného uzlu normalizováni. Následuje pseudokód 2 definující právě tento prefixový průchod.

6.3 Implementace normalizačních pravidel

Některá normalizační pravidla vyžadují provedení transformačních úkonů ve stromové struktuře. Nejprve si projdeme pravidla, která vyžadují transformace.

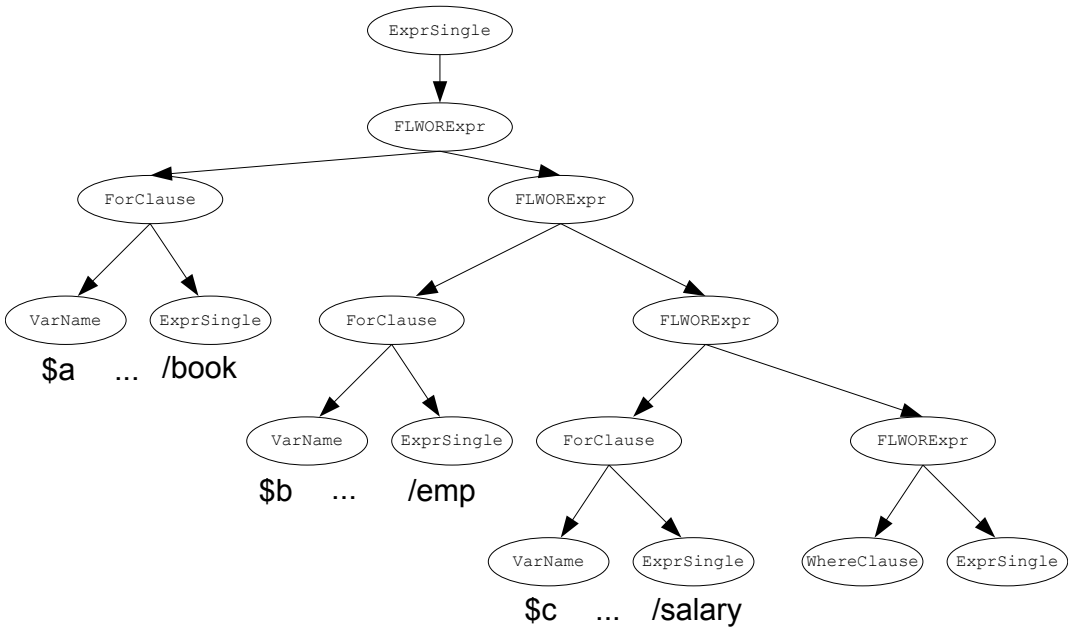
6.3.1 FOR transformace

Již víme, že složený konstrukt FOR se rozkládá na jednotlivé FOR konstrukty svázané vždy pouze s jednou proměnnou. Tento implementační problém je řešen pomocí transformace stromu. Model původního výrazu **for $a in /book, $b in /emp, $c in /salary where ...**

```
1 if simpleNode je určen k normalizaci then
2 |  Normalizuj simpleNode podle typu uzlu;
3 else
4 |  foreach child do
5 |  |  Vykonej tuto rekurzivní metodu normalize(child);
6 |  end
7 end
```

Algoritmus 2: Prefixový průchod

je zobrazen i s první transformací již dříve na obrázku 13. Výsledná stromová struktura je zobrazena na obrázku 14. Abychom obrázek, co nejvíce zjednodušili, tak byly vypuštěny volitelné části, jako jsou určení datového typu, či výčtová hodnota.



Obrázek 14: Normalizovaný výraz **for $a in /book, $b in /emp, $c in /salary where ...**

V následujícím výpise je zobrazen normalizovaný výstup konstruktu FOR. Konstrukt WHERE je popsán později 6.3.3. Výstupní výraz tedy, bude vypadat takto:

```
for $a in /book return
    for $b in /emp return
        for $c in /salary return
            ...
```

Výpis 16: Normalizovaný výstup XQuery výrazu

Pseudokód pro normalizaci FOR a LET pravidel Pseudokód 3 je společný pro obě pravidla. Jediný rozdíl je ve volitelných částech. FOR může obsahovat výčtovou hodnotu a určení datového typu. LET umožňuje pouze určení datového typu.

```
1  if simpleNode je "FORClause" nebo "LETClause" then
2  |  Transformuj uzel simpleNode
3  else
4  |  Procházej rekurzivně dětské uzly simpleNode
5  end
6  Nyní transformujeme uzly;
7  foreach child do
8  |  if child je typu "ExprSingle" then
9  |  |  Vytvoření nového rodiče newParent, který bude typu "FLWORExpr";
10 |  |  Vytvoření nového dítěte newChild, které bude typu "FORClause" případně
   |  |  "LETClause";
11 |  |  Nastavení vazeb - newParent je rodič pro newChild a naopak;
12 |  |  for Projdi všechny následující dětské uzly child do
13 |  |  |  Přesuň následující child pod newChild
14 |  |  end
15 |  |  newParent je nastaven jako dítě parent;
16 |  |  foreach nextParentChild - osa following-sibling do
17 |  |  |  Přidej nextParentChild jako dětské uzly pod newParent a nastav
   |  |  |  vazby - tento přepis je nutný pro kombinace FOR a LET;
18 |  |  end
19 |  end
20 end
```

Algoritmus 3: FOR a LET normalizace

6.3.2 LET transformace a kombinace s konstruktem FOR

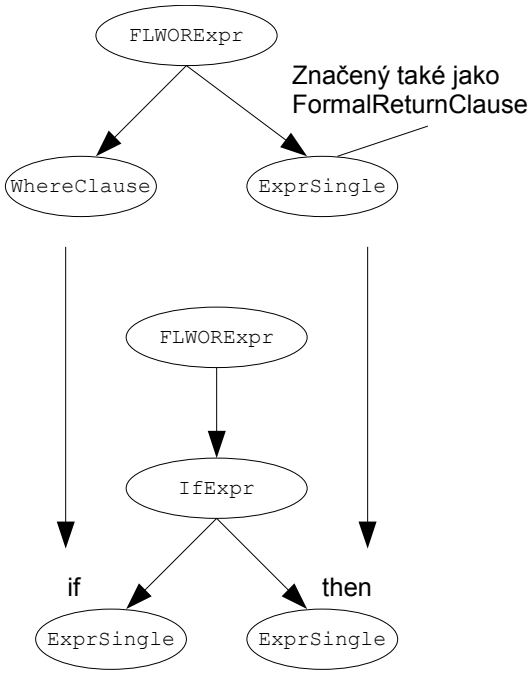
Transformace konstruktu LET je obdobná jako transformace konstruktu FOR. Nyní si ukážeme, jak probíhá transformace, když jsou tyto konstrukty mezi sebou kombinovány. Algoritmus 3 znázorňuje pseudokód pro normalizaci FOR a LET konstruktů. U těchto konstruktů se vykonávají dva druhy přesouvání. Nejprve přesuny dětských uzlů, které jsou pod $simpleNode$, což může být složený FOR nebo LET. Druhý přesun se týká následujících uzlů, které jsou na stejné úrovni ,jsou sousedé, jako právě transformovaný uzel. Tyto uzly se přesunou za $simpleNode$, jako další díte $newParent$.

6.3.3 WHERE normalizace

Normalizace konstruktu WHERE (obrázek 15) se provádí na konstrukci **If-Then-Else**, ovšem Else neobsahuje žádnou funkcionalitu. Pseudokód pro WHERE transformaci je zapsán ve 4. výpise zdrojového kódu.

```
1  simpleNode - "WhereClause";
2  parent - "FLWORExpr";
3  Vytvoření nového uzlu IfExpr, který bude potomkem parent;
4  foreach parentChild do
5      if parentChild je původní "WHEREClause" then
6          Nahraď "IfExpr" za "WHEREClause";
7          nextParentChild = následující uzel - jedná se o "FormalReturnClause";
8      end
9  end
10 Přidej pod uzel "IfExpr" původní "WHEREClause" a nextParentChild, což je
   "FormalReturnClause";
```

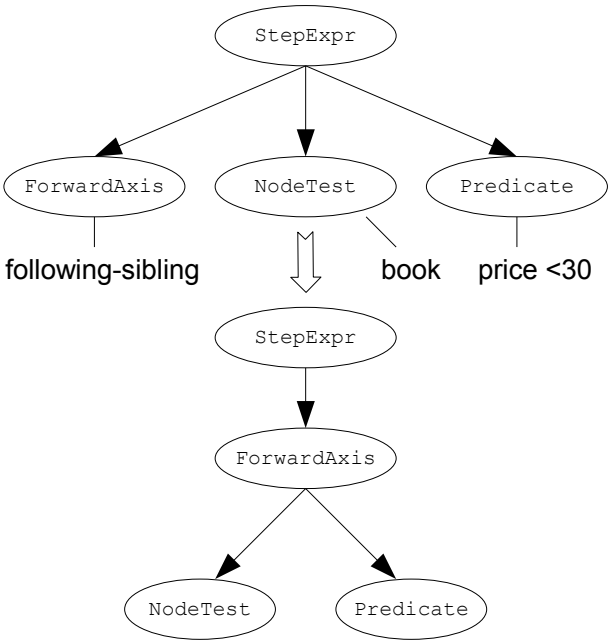
Algoritmus 4: WHERE normalizace



Obrázek 15: WHERE transformace na IF

6.3.4 Transformace os

Transformace os je velmi triviální. Jedná se pouze o přesun uzlů následujících za aktuálně zpracovávaným uzlem popisující osu. Například transformace pro výraz $/bookstore/following - sibling :: book[price < 30]$, je zobrazena na obrázku 16. Tento způsob velmi usnadňuje následné generování výpisu pro osy, protože dokážeme lehce rozpoznat uzly, které souvisí s touto osou.



Obrázek 16: Transformace osy

```
1 simpleNode - "ForwardAxis" nebo "ReverseAxis";
2 parent - "StepExpr";
3 foreach parentChild do
4     if parentChild je původní "simpleNode" then
5         nextChildIndex je index následujícího uzlu;
6         break;
7     end
8 end
9 Přesunutí všech následujících uzlů pod "simpleNode";
```

Algoritmus 5: Transformace os

6.4 Generování výstupu

Pokud jsme prefixovým průchodem prošli celý strom, následuje další fáze a to zpracování stromu zpět do XQuery výrazu. Ovšem tento výraz, na rozdíl od původního, již využívá XQuery Core gramatiku. To znamená, že neobsahuje redundantní výrazy. Generování výpisu je implementováno metodou *print* ve třídě *Normalization*. Metoda *print* využívá rekurzivní procházení stromu, díky kterému projdeme celý strom a postupně vypisujeme jednotlivé části XQuery výrazu v závislosti na aktuálně zpracovávaném uzlu. V následujícím pseudokódu 6 je popsáno generování výpisu pro uzel *ForClause*, který reprezentuje konstrukt FOR.

```
1 simpleNode - "ForClause";
2 output - Generovaný výstup;
3 if simpleNode je "ForClause" then
4     output přidej "for ";
5     Zpracuj prvního potomka simpleNode;
6 end
7 if nextChild reprezentuje volitelný neterminál "TypeDeclaration" nebo "PositionalVar"
  then
8     Zpracuj tyto volitelné neterminály (přidání "at " nebo "as " do output);
9 end
10 output přidej "in ";
11 Zpracuj další nextChild;
12 output přidej "return ";
```

Algoritmus 6: Generování výpisu pro konstrukt FOR

6.5 Implementovaná pravidla

Nyní se zmíníme o normalizačních pravidlech, které byly implementovány v této práci. Podrobnější informace o těchto pravidlech jsou rozepsány v kapitole 4.

- FLWOR normalizace (mimo konstrukt ORDER BY - ten je při generování výstupu vypuštěn).
- Normalizace cest.
- Normalizace os a jejich zkrácené zápisy.
- Určení kontextu.
- Rozsahový operátor **to**.

7 Analyzování výstupních dat

V této kapitole porovnáme dva různé XQuery výrazy, které se normalizují na naprosto stejný výraz. A také si popíšeme možnost podrobného výpisu do souboru **log.txt**.

7.1 Normalizování redundantních výrazů

Nyní si ukážeme příklad dvou XQuery výrazů, které jsou normalizovány na stejný výstupní výraz.

```
for $i in $I, $j in $J let $k := $i + $j, $l := $L where $m >= 5 return ($i,$j)

for $i in $I for $j in $J let $k := $i + $j let $l := $L return if ($m >= 5) then ($i,$j) else ()
```

Výpis 17: Normalizace redundantních XQuery výrazů

Tyto výrazy jsou normalizovány na tento výstupní výraz:

```
for $i in $I return
  for $j in (fn:root(self::node()) treat as document−node()) /descendant−or−self:: book return
      let $k := $i +$j return
          let $l := $L return
              if (fn:boolean ($m >=5 )) then ($i ,$j ) else ()
```

Výpis 18: Výstupní normalizovaný výraz

7.2 Detaily normalizace

V případě, že si chceme prohlédnout detailněji průběh normalizace a stromovou strukturu před a po normalizaci, tak byla doimplementována funkce, která podle zadaného parametru vypíše pomocné informace do souboru **log.txt**. Existují 2 typy výpisu:

- **-o** - Vypíše pouze vykonané transformace pro daný výraz.

```
Vykonané transformace:

For transformace.
For transformace.
Transformace dopředné osy.
Let transformace.
Let transformace.
```

Výpis 19: Transformační výpis v souboru **log.txt**

- **-a** - Mimo použité transformace se vypíše také stromová struktura před a po normalizaci.

```
Výpis stromu před normalizací:

|Uzel [0]  Module
 |Uzel [0]  MainModule
  |Uzel [0]  Prolog
Prolog  nemá žádného potomka!!!
  |Uzel [1]  QueryBody
   |Uzel [0]  Expr
...

Vykonané transformace:
...

Výpis stromu po normalizaci:

|Uzel [0]  Module
 |Uzel [0]  MainModule
  |Uzel [0]  Prolog
Prolog  nemá žádného potomka!!!
...
```

Výpis 20: Podrobný transformační výpis v souboru **log.txt** pro FLWOR konstrukci

8 Závěr

Výsledná aplikace splňuje všechny požadavky zadání. Podařilo se nám naimplementovat nejkomplikovanější normalizační pravidla, která byla odsouhlasena na konzultacích. Předpokladem úspěšného implementování normalizačních pravidel bylo nastudování jazyka XQuery s důrazem na jeho formální sémantiku. Dále práce s parser generátory a v neposlední řadě transformace modelu XQuery výrazu, který byl reprezentován stromovou strukturou.

Program byl vytvořen v programovacím jazyce Java, protože využívá JJTree soubor, který je určen pro práci s tímto jazykem. V programu se odráží má snaha o co nejlepší ukázku dané problematiky zaměřena na FLWOR konstrukce, které jsou stěžejní pilíře jazyka XQuery. Výsledný program byl otestován a několik jednodušších příkladů je zobrazeno v kapitole o normalizaci XQuery. V kapitole zabývající se analýzou normalizace jsou zobrazeny dva XQuery výrazy, které jsou normalizovány na stejný výstupní výraz.

Tato práce má za cíl nastínit problematiku aplikování normalizačních pravidel na celou sadu XQuery výrazů, než-li se stát plnohodnotným programem zajišťujícím převod do normalizované formy. Obsahuje teorii k pochopení základní problematiky normalizačních pravidel, včetně jejich obecného zápisu.

Jako další vylepšení programu bych viděl v implementaci zbývajících normalizačních pravidel, například kvantifikátory nebo ORDER BY konstrukt. Nicméně u takovýchto pravidel se spíše jedná o zdlouhavý procedurální výpis, než o pravidlo vedoucí ke změně struktury výrazu.

9 Literatura

[1] *Extensible Markup Language* [online]. [cit. 17.4.2010]. Dostupné z : <http://cs.wikipedia.org/wiki/Extensible_Markup_Language>.

[2] KOSEK, Jiří. *XML* [online]. [cit. 17.4.2010]. Dostupné z : <http://www.kosek.cz/clanky/xml/xml-uvod.html>.

[3] W3C. *Extensible Markup Language (XML)* [online]. [cit. 17.4.2010]. Dostupné z : <http://www.w3.org/XML/>.

[4] KOSEK, Jiří. *XML schémata* [online]. [cit. 17.4.2010]. Dostupné z : <http://www.kosek.cz/xml/schema/>.

[5] BŘÍZA, Petr. *Základy jazyka XPath* [online]. [cit. 17.4.2010]. Dostupné z : <http://interval.cz/clanky/zaklady-jazyka-xpath/>.

[6] HERNANDEZ, George. *XPath* [online]. [cit. 17.4.2010]. Dostupné z : <http://www.georgehernandez.com/h/xComputers/XML/XSL/XPath.asp>.

[7] NIČ, Miloslav. *XPath* [online]. [cit. 17.4.2010]. Dostupné z : <http://www.zvon.org/xxl/XPathTutorial/General_cze/examples.html>.

[8] HAPL, Martin. *Úvod do XSLT* [online]. [cit. 17.4.2010]. Dostupné z : <http://www.biz-portal.net/xslt-uvod>.

[9] SPURNÝ, Lukáš. *Saxon* [online]. [cit. 17.4.2010]. Dostupné z : <http://www.abclinuxu.cz/software/nastroje/saxon>.

[10] KAY, Michael. *SAXON - The XSLT and XQuery Processor* [online]. [cit. 17.4.2010]. Dostupné z : <http://saxon.sourceforge.net/>.

[11] *Deklarativní programování* [online]. [cit. 17.4.2010]. Dostupné z : <http://cs.wikipedia.org/wiki/Deklarativn%C3%AD_programov%C3%A1n%C3%AD>.

[12] W3C. *XQuery 1.0: An XML Query Language* [online]. [cit. 17.4.2010]. Dostupné z : <http://www.w3.org/TR/xquery/>.

[13] SIMEON, Jerome. *XQuery from the Experts*. Boston : Pearson Education, Inc., 2004.

[14] SCHWARTZBACH, Michael. *FLWOR expressions*. [online]. [cit. 17.4.2010]. Dostupné z : <http://www.brics.dk/ amoeller/XML/querying/flwrexp.html>.

[15] W3C. *XQuery 1.0 and XPath 2.0 Formal Semantics*. [online]. [cit. 17.4.2010]. Dostupné z : <http://www.w3.org/TR/xquery-semantics/>.

[16] *Bubble sort*. [online]. [cit. 17.4.2010]. Dostupné z : <http://www.algoritmy.net/article/3/Bubble-sort>.

[17] VYSKOČIL, Michal. *Jazyky a překladače*. [online]. [cit. 17.4.2010]. Dostupné z : <http://www.abclinuxu.cz/serialy/jazyky-a-prekladace>.

[18] *Parser*. [online]. [cit. 17.4.2010]. Dostupné z : <http://cs.wikipedia.org/wiki/Parser>.

[19] *Lexikální analýza*. [online]. [cit. 17.4.2010]. Dostupné z : <http://cs.wikipedia.org/wiki/Lexik%C3%A1ln%C3%AD_anal%C3%BDza>.

[20] *JavaCC*. [online]. [cit. 17.4.2010]. Dostupné z : <https://javacc.dev.java.net/>.

[21] KATZ, Howard. *JavaCC*. [online]. [cit. 17.4.2010]. Dostupné z : <http://www.ibm.com/developerworks/xml/library/x-javacc1.html>.

[22] PAINE, Jocelyn. *Introduction to JJTree*. [online]. [cit. 17.4.2010]. Dostupné z : <http://www.j-paine.org/jjtree.html>.

[23] *LL syntaktický analyzátor*. [online]. [cit. 17.4.2010]. Dostupné z : <http://cs.wikipedia.org/wiki/LL_syntaktick%C3%BD_analyz%C3%A1tor>.

[24] *Překladač*. [online]. [cit. 17.4.2010]. Dostupné z : <http://cs.wikipedia.org/wiki/P%C5%99eklada%C4%8D>.

[25] *JavaCC Eclipse Plug-in*. [online]. [cit. 17.4.2010]. Dostupné z : <http://eclipse-javacc.sourceforge.net/>.

[26] KOTALA, Zdeněk; TOMAN, Petr. *Java sborník* [online]. [cit. 23.3.2010]. Dostupné z : <http://dione.zcu.cz/java/sbornik/toc.html>.

[27] HEROUT, Pavel. *Učebnice jazyka Java*. České Budějovice : Kopp, 2004.

[28] *Java Platform SE 6* [online]. [cit. 26.3.2010]. Dostupné z : <http://java.sun.com/javase/6/docs/api/>.

[29] *XQuery Update Facility 1.0* [online]. [cit. 26.4.2010]. Dostupné z : <http://www.w3.org/TR/xquery-update-10/>.

[30] *Extended Backus–Naur Form* [online]. [cit. 28.4.2010]. Dostupné z : <http://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_Form>.

A Obsluha programu

Program byl vytvořen v programovacím jazyku Java JDK 1.6. Spustit program lze dvěma způsoby:

1. Pomocí jar souboru - Do konzole najedeme do adresáře s jar souborem. Poté příkazem **java -jar xquery.jar "XQuery"** spustíme normalizaci. "XQuery" je vstupní výraz, který chceme normalizovat.
2. Pomocí Eclipse - Spustím **Run Configurations** a nastavím si parametry. Poté již stačí Run.

B Programová specifikace

B.1 Vývojové prostředí

Pro psaní programů je k dispozici celá řada vývojových prostředí (IDE), a to jak freewarových, tak i komerčních. Tento program byl vytvořen ve vývojovém prostředí Eclipse 3.4.0., který je zdarma ke stáhnutí na http://www.eclipse.org/. Toto vývojové prostředí se nám velmi osvědčilo také díky možnosti využití JavaCC pluginu, který zajišťuje překlad z JavaCC a JJTree souborů do java tříd.

B.2 Latex

Teoretická část diplomové práce byla vypracována v profesionálním sazbovém prostředí Tex, konkrétně nástavby LaTex 2007 s podporou českého jazyka balíkem CSLaTeX. Pro grafické rozhraní LaTexu byl použit editor TeXnicCenter.

B.3 Diagramy

Všechny vytvořené diagramy byly vytvořeny pomocí programu Draw z kancelářského balíku OpenOffice.org. Vzorce a zápis normalizačních pravidel byl vytvořen v programu Math z téhož kancelářského balíku.